Unloading Java Classes That Contain Static Fields

C. E. McDowell
E. A. Baldwin
Computer Science Dept., University of California
Santa Cruz, CA 95064
email: charlie@cs.ucsc.edu, phone: (408) 459-4772

October 15, 1997

Abstract

In Java, the definition of a "program" is a bit fuzzy. A Java applet is essentially a Java application (i.e. program) that can be executed by a Java enabled Web browser (i.e. an OS). An applet running inside of a browser was intended to be analogous to a conventional application running under on OS, hence the netcentric "browser is your OS" model. However, as currently implemented, this analogy breaks down with regard to the system resources allocated for classes and in particular for static fields in classes (i.e. class variables) when the class was loaded as part of an applet.

Without class unloading, a long running Java application such as a browser is like an OS that does not release memory resources allocated for application code space when the application terminates. With class unloading, as currently implemented, the semantics of static fields in classes are broken. In this paper we detail the problem and provide a solution. The solution combines restricting when classes can be unloaded, with a greater use of non-default class loaders.

Keywords: Java, class unloading, garbage collection

1 Introduction

Two of the features of the Java programming language that have contributed to its popularity are garbage collection of heap allocated data, and dynamic linking and loading of class libraries. A major target application for Java is in Web browsers, with the network plus Java and its APIs replacing the conventional operating system.

In conventional operating systems today, there is generally a very clear distinction between the long lived program or programs that are the OS, and an application that is executed by the OS. Some resources, such as files, may have persistence across application executions, but any main memory resources can be recovered when the application "exits." In Java, the definition of a "program" is a bit fuzzy. A Java applet is essentially a Java application (i.e. program) that can be executed by a Java enabled Web browser (i.e. an OS). An applet running inside of a browser was intended to be analogous to a conventional application running under on OS, hence the netcentric "browser is your OS" model. However, as currently implemented, this analogy breaks down with regard to the system resources allocated for classes, and in particular for static fields in classes (i.e. class variables) when the class was loaded as part of an applet.

The Java Language Specification (JLS)[GJS96] allows for the unloading of classes. Without class unloading, a Java Virtual Machine (JVM) is like an OS that never releases the memory allocated for the code space of an application, even after the application has completed. In particular the JLS states that "This can be used, for example, to unload a group of related types... Such a group might consist of all the classes implementing a single applet." Class unloading is important for any long running Java program that continuously loads classes that are used for some time and then no longer used. This is exactly the

type of behavior that occurs with applets. Applets are Java classes that are intended to be loaded into the Java Virtual Machine of a WWW browser when the browser encounters the appropriate HTML tag. One of the features of Java that makes it attractive for many applications is the security model that makes this dynamic loading and execution of untrusted program fragments possible. In the "network is the computer" model where a browser or similar Java program is your operating system interface, it is essential that classes be unloaded periodically. Otherwise the computer's memory will become cluttered with classes that are no longer being used. This is marginally tolerable in a system with virtual memory or one in which the JVM is frequently restarted. For long running or embedded systems without virtual memory, class unloading is necessary if there are any dynamically loaded classes.

An undesirable (or at least potentially unexpected) result of class unloading as implemented in JDK1.1.4, the free Java system from JavaSoft, is that a static field in a class can get reinitialized. This is in direct conflict with section 8.3.1.1 of the Java Language Specification which states that for static fields "there exists exactly one incarnation of the field, no matter how many instances (possibly zero) of the class may eventually be created." Unfortunately this is not simply an error in the JDK1.1.4 implementation of the JVM, in fact JavaSoft may not even consider it a bug. It is certainly an ambiguity in the JLS and it raises the question of how you can provide class unloading in such a way as to preserve the expected behavior for static fields in classes.

2 When does the problem arise?

There are two general ways a class can get loaded. The most common way is for the class name, call it ClassB, to actually appear in some other class, call it ClassA. The class won't actually get loaded until the program performs some action that requires ClassB to be loaded[1], e.g. creates an instance or references a static member or field of ClassB. The other way for a class to be loaded is some form of loading where the class name only appears as a String such as with the method `forname` in class `java.lang.Class` (see JLS section 20.3.8[GJS96]), or the use of method `loadClass` from a custom class loader extending class `ClassLoader`.

According to the JLS section 12.8, the only restrictions on class unloading are:

1. "A class may not be unloaded while any instance of it is still reachable."
2. "A class or interface may not be unloaded while the Class object that represents it is still reachable."

There appears to be some ambiguity as to when a `Class` object is reachable. One way for a `Class` object to be reachable is if there is a normal Java reference to the `Class` object, e.g. a variable of type `Class` that references the class in question. But a class may also be considered reachable when there are no explicit `Class` type variables referencing the `Class` object. In our implementation of class unloading which we added to JDK1.0.1[Bal97, Mon97], ClassB is directly reachable from ClassA, if the source for ClassA contains a reference to ClassB as a class. This is not a variable of type `Class` but the actual use of the class name, for example in a declaration. Specifically we look for an occurrence of ClassB as a class value in the constant pool for ClassA. The constant pool is the part of the class file format that stores references to other classes. For unresolved classes the constant pool will contain a string that is the actual class name associated with a tag indicating this is a class name. For resolved classes the constant pool will contain a direct reference to the `Class` object.

If a class was loaded using the `forname()` method from class `Class` or `loadClass()` from `ClassLoader`, then in JDK1.1.4 the `Class` object is no longer reachable when all references to the object returned by the call to `forname()` have been removed. In contrast, if the class was loaded by explicit reference to the class name as a type, then the class will continue to be reachable as long as the class containing the explicit reference is reachable. The class will be reachable even when all instances of the class have been made unreachable and no user visible variables of type `Class` reference the class. This is actually consistent with our own implementation of class unloading. We consider a class to be reachable (i.e. not unloadable) if there are instances of the class or if there are reachable classes that contain resolved or *unresolved* references to the class. Including unresolved references as reachable classes is different from the behavior we observed

[1] The actual loading of ClassB could happen anytime but if there is a problem with the loading of ClassB then that error cannot be reported until the action in ClassA occurs that would require ClassB.

in JDK1.1.4. Unfortunately, even using our definition of reachable, it is possible to have a class become unreachable, and then later become reachable again. The class that becomes unreachable and then later reachable does not need to have been loaded using `forname()`, although it would appear that the class must have been loaded indirectly as the result of a call to `forname()` (see the example in the following section). A solution we will describe below is to add an additional requirement for class unloading - *do not unload a class if the class loader that loaded the class is still reachable and the class contains static fields.* An unfortunate result of this restriction is that no classes with static fields, loaded by the default class loader, will ever be unloaded.

The Java language semantics require that errors related to class loading not be reported until some program activity that required the class to be loaded occurs (JLS 12.2.1). This means it is not possible to analyze a complete program at load time to determine which classes might appear to be unloadable but then get reloaded later on.

3 An example

In JDK1.1.4, class unloading is done at the same time as garbage collection, although this may not be the best policy [Bal97]. The following example demonstrates how both a class that is loaded by a call to `forname` (`ClassOne` in the example) and a class that is loaded by an explicit use of the class as a type (`ClassTwo` in the example) are unloaded. In addition there is an explicit use of `ClassOne` as a type in `main` yet the class still gets unloaded by JDK1.1.4. `ClassOne` would not be unloaded using our definition of *reachable* because the constant pool for the class `Example` contains a reference to `ClassOne`.

A strict interpretation of the JLS concerning static fields would prevent any unloading of a class with static variables if the class was loaded by the default class loader. We believe that the loader that loaded a class must also be unreachable before a class can be unloaded. The following code example demonstrates that Sun does not make a strict interpretation of this in JDK1.1.4. In this example, the creation of the object first assigned to `class_one_object` (actually an instance of `ClassOne`) also creates an instance of `ClassTwo` which contains a static field. Once the references to the `ClassOne` object and the `Class` object are set to null, a call to the garbage collector results in both `ClassOne` and `ClassTwo` being unloaded. This can be seen because when the final assignment to `class_one_object` occurs, creating a new instance of `ClassOne` and a new instance of `ClassTwo`, the program will print out the value of counter in `ClassTwo` as 1 indicating it was reinitialized to 0, clearly contradicting the language specification with regard to static fields. We believe the program should not unload `ClassTwo` in this example. If the class is not unloaded then the program will print out 1 and then 2, which it does if the call to the garbage collector, and hence class unloading, is removed. Another disturbing aspect of this program is that it is clearly non-deterministic with respect to when garbage collection occurs.

One might be tempted to simply dismiss this as a bug in JDK1.1.4, which it may well be, however, this example clearly illustrates a problem with class unloading. What is the right thing to do? Can you ever unload a class with a static field? If not, this might very well stop the unloading of most classes that we would like to unload. The class unloading we added to JDK1.0.1 in our experimental system exhibits similar problems, although it would not unload any classes in this example. In our system, if the last line in the example is replaced with a call to `Class.forName` and the instance is created with a call to `newInstance` as was done earlier in the example, then both `ClassOne` and `ClassTwo` would be unloaded. This change removes the reference to `ClassOne` from the constant pool of the class `Example`.

In the remainder of this paper we will explore some possible solutions that may be preferable to disallowing unloading of a class with static fields.

```
class Example
{
    public static void main(String[] args)
    throws java.io.IOException, ClassNotFoundException,
    IllegalAccessException, InstantiationException
    {
        int count = 0;
        SomeInterface class_one_object;

        /* load the class ClassOne and instantiate an instance of it */
        /* ClassOne uses ClassTwo which will thus get loaded also */
        Class theClass = Class.forName("ClassOne");
        class_one_object = (SomeInterface)theClass.newInstance();

        /* remove all references to the class and the instance */
        class_one_object = null;
        theClass = null;

        System.gc(); /* force garbage collection which also unloads classes */

        /*loads and instantiates ClassOne again. ClassTwo also gets reloaded*/
        class_one_object = (SomeInterface)new ClassOne();
    }
}

public class ClassOne implements SomeInterface{
    public ClassOne(){
        new ClassTwo();
    }
}

public interface SomeInterface {
}

public class ClassTwo {
    static int counter=0;
    public ClassTwo(){
        counter++;
        System.out.println("ClassTwo has counter = " + counter);
    }
}
```

4 Using class loaders to limit class lifetimes

The problem as described in the introduction is that there needs to be some clear boundary marking the beginning and the end of an "application" running on top of some "OS" running on top of a JVM. For applets there is the constructor that creates the applet to mark the beginning and a destroy method that gets called before the applet is eliminated. This delimits the applet and could be used to reclaim some resources. This is not so much the applet saying it is done as the OS telling the applet that it is done. The question again is, can you safely unload the classes used by the applet without violating the semantics of static fields? As described below, the answer is no for existing systems. The alternative we propose in this section is to disallow any unloading of system classes (i.e. any classes loaded by the default class loader) and

use a separate class loader for each "application" that may load classes that will need to be unloaded.

Java provides a mechanism for a Java program to load classes under the control of a custom class loader. Two classes with the same name and identical class files will be considered different to a JVM if loaded using different class loaders, even if the class loaders are simply two instances of the same class loader class. Once the class loader object is no longer reachable, then the class unloading policy stated in the JLS can safely be applied (i.e. no instances of the class exist and the class object itself is not reachable). It would be impossible for the same class to be reloaded because "sameness" includes having been loaded by the same class loader which is no longer accessible.

Although current browsers, to our knowledge, do not yet implement class unloading, they do allocate one class loader for each distinct applet code base. In this way, if the browser determines, using any criteria it wishes, to terminate the execution of all applets loaded from a particular code base, then those classes loaded via that class loader can safely be unloaded. Specifically, it would be impossible for an applet to detect the reloading and consequent reinitialization of a static field without resorting to external, persistent resources.

Based on our proposal, no system classes containing static fields will be unloaded. This is not how JDK1.1.4 operates. It is clearly possible to have "system" classes (i.e. those found in the CLASSPATH) unloaded, as demonstrated in the example earlier. For systems with large amounts of memory or with support for virtual memory, having most or all system classes loaded during steady state may not be a problem. This restriction might be undesirable for memory limited embedded systems.

If it was necessary to unload system classes, then the "OS" could split the true system classes from the other classes that would currently be loaded by the default class loader. By true system class we mean one that must be loaded by the default class loader for security reasons. These later classes could then be loaded by the class loader used to load the actual application (e.g. the applet).

This use of class loaders would allow a programmer to write a Java program that functioned as the user interface to the "OS" (i.e. the one running on top of the JVM) and allowed for unloading of all classes loaded by an application when the "OS" detected that the application had terminated.

5 Conclusion

In this paper we have identified a problem with the Java Language Specification with regard to class unloading and static fields. We have provided an example that demonstrates how the current implementation of class unloading can result in a static field being reinitialized resulting in a program that changes its behavior depending upon when garbage collection occurs. We then describe a possible solution, specifically, a class with static fields should not be unloaded until the class loader that loaded the class is no longer reachable.

References

[Bal97] E. Baldwin. *Memory Management in Embedded Java.* M.S. Thesis University of California, Santa Cruz, 1997.

[GJS96] J. Gosling, B. Joy, and G. Steele. *The Java Language Specification.* Addison Wesley, 1996.

[Mon97] B. R. Montague. JN: OS for an embedded java network computer. *IEEE Micro*, 17(3):54–60, May/June 1997.